

# PROYECTO PARQUE SALITRE MAGICO

**Elaborado por:**

Nelbis Maria Olivero Rondon

## RETO TÉCNICO

Proyecto de la empresa "Parque Salitre Mágico", un parque de atracciones.

El reto consiste en desarrollar una aplicación MVC que integre un CRUD apoyándose en los conocimientos de POO, manejo de GIT, conocimientos de bases de datos relacionales y/o no relacionales, que permita conocer al parque información sobre el uso de sus atracciones.

El parque cuenta con tres tipos de entrada (pasaportes) y la estatura es una condición que le permite ingresar a las atracciones o no.

El reto es una oportunidad para que las personas demuestren sus habilidades y conocimientos en tecnologías de desarrollo de software.

### 1. Tecnologías Reto:

La aplicación deberá ser desarrollada e implementada en las siguientes tecnologías:

- ✓ Base de datos relacional (Postgresql, sqlite, mysql) o no relacional
- ✓ (mongoDB, redis) que le permita tener al sistema persistencia.
- ✓ POO en cualquier lenguaje de programación preferiblemente Java.
- ✓ Manejo de arreglos o colecciones.
- ✓ GIT y Github.
- ✓ La aplicación puede ser ejecutada por consola, interfaz gráfica de escritorio o
- ✓ Web (JS).

Teniendo en cuenta el contexto anterior, deberás realizar lo siguiente:

### 2. Funcionalidades Parque de Diversiones:

1. Cada persona deberá ser un cliente del sistema independiente de la edad que tenga, después de haber ingresado la primera vez al parque y de llenar un

formulario con la información básica suficiente que permita enviarle promociones a sus mejores clientes (frecuentes), la información mínima requerida es nombre, cédula, teléfono, correo electrónico, estatura, edad, en caso de ser menor de edad debe tener asociado información de contacto de un familiar, el departamento de publicidad no podrá enviarle información publicitaria o promociones.

2. El parque tiene 5 tipos de empleados (administrativos, logística, publicidad, operadores y mantenimiento) de los cuales se tiene como mínimo información personal y horario laboral, los operadores son los encargados de verificar la estatura y el tiquete y con esto permitir el ingreso a las atracciones.
3. Los tiquetes son adquiridos en 5 estaciones diferentes dentro del parque y son entregados por el personal de logística. Según el día de la semana y la temporada las estaciones pueden estar habilitadas algunas o todas según el porcentaje de ocupación, la habilitación de una estación será responsabilidad del personal administrativo.
4. Aunque el cliente tenga acceso a una atracción y cumpla con los requisitos para ingresar, se debe validar que la máquina no esté inhabilitada por daño o mantenimiento, la información del estado de una máquina “no disponible” debe ser de un empleado de mantenimiento y debe ser informado a las 5 estaciones.
5. Adicional a todo lo mencionado antes las atracciones deben tener al menos una descripción, una clasificación y unas condiciones de uso que pueden ser modificadas por un empleado administrativo.
6. Un usuario que no tenga la estatura requerida para cada atracción no podrá hacer uso de la misma y se debe mostrar alerta a los operadores.
7. El parque desea conocer la cantidad de personas que se encuentran en su interior en cualquier momento para poder determinar su porcentaje de ocupación y poder habilitar las estaciones necesarias, agregue lo que considere necesario para poder visualizar esta información.
8. El departamento administrativo quiere conocer las atracciones que los clientes

prefieren, y las atracciones menos visitadas.

9. El departamento de publicidad quiere conocer qué clientes visitan el parque con mayor frecuencia para determinar una promoción personalizada, por ejemplo, un cliente que visita 3 veces el parque tendrá un xx % de descuento para su próxima visita.

## ANALISIS Y SOLUCIÓN

Nuestra aplicación será desarrollada e implementada con las siguientes tecnologías:

- ✓ Base de datos relacional con MySQL.
- ✓ POO en lenguaje Java.

La estructura de dicho proyecto es el siguiente:

### Estructura del Proyecto

1. **src**
  - **main**
    - **java**
      - **com**
        - **parque**
          - **modelo**
            - Cliente.java
            - Empleado.java
            - Atraccion.java
            - Tiquete.java
            - Estacion.java
          - **controlador**
            - ControladorParque.java
          - **vista**
            - VistaConsola.java
          - **ParqueSalitreMagico.java**
          - **db**
            - DatabaseConnection.java

Esta estructura cumple con el patrón de diseño de software MVC ( Modelo-Vista – Controlador), está organizada en paquetes que representan diferentes aspectos de la aplicación. A continuación, se detalla la estructura y la funcionalidad de cada componente:

**Paquete Modelo:** Se encarga de los datos y la lógica de negocios.

Código de clases:

### 1. Cliente:

```
package com.salitreparkemagicoapp.modelo;

public class Cliente {
    private String nombre;
    private String cedula;
    private String telefono;
    private String correo;
    private double estatura;
    private int edad;
    private String contactoFamiliar; // Solo se usará si el cliente es
menor de edad
    private int visitas; // Contador de visitas al parque
    private boolean esFrecuente; // Indica si el cliente es frecuente
    private boolean puedeRecibirPromociones; // Indica si el cliente
puede recibir promociones

    // Constructor
    public Cliente(String nombre, String cedula, String telefono, String
correo, double estatura, int edad, String contactoFamiliar, int visitas)
    {
        this.nombre = nombre;
        this.cedula = cedula;
        this.telefono = telefono;
        this.correo = correo;
        this.estatura = estatura;
        this.edad = edad;
        this.contactoFamiliar = (edad < 18) ? contactoFamiliar : null; //
Asocia contacto familiar solo si es menor de edad
        this.visitas = visitas;
        this.esFrecuente = visitas >= 5; // Considerar frecuente si ha
visitado 5 veces o más
        this.puedeRecibirPromociones = edad >= 18; // Solo mayores de
edad pueden recibir promociones
    }

    // Getters y Setters
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getCedula() {
        return cedula;
    }
}
```

```

    }

    public void setCedula(String cedula) {
        this.cedula = cedula;
    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public String getCorreo() {
        return correo;
    }

    public void setCorreo(String correo) {
        this.correo = correo;
    }

    public double getEstatura() {
        return estatura;
    }

    public void setEstatura(double estatura) {
        this.estatura = estatura;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
        this.puedeRecibirPromociones = edad >= 18; // Actualiza si puede
recibir promociones
        if (edad < 18) {
            this.contactoFamiliar = contactoFamiliar; // Mantiene el
contacto familiar si es menor
        } else {
            this.contactoFamiliar = null; // No necesita contacto
familiar si es mayor
        }
    }

    public String getContactoFamiliar() {
        return contactoFamiliar;
    }

    public void setContactoFamiliar(String contactoFamiliar) {
        this.contactoFamiliar = contactoFamiliar;
    }

```

```

    public int getVisitas() {
        return visitas;
    }

    public void setVisitas(int visitas) {
        this.visitas = visitas;
        this.esFrecuente = visitas >= 5; // Actualiza el estado de
cliente frecuente
    }

    public boolean isEsFrecuente() {
        return esFrecuente;
    }

    public boolean setPuedeRecibirPromociones() {
        return puedeRecibirPromociones;
    }

    // Método para establecer el estado de promociones (opcional)
    public void setPuedeRecibirPromociones(boolean
puedeRecibirPromociones) {
        this.puedeRecibirPromociones = puedeRecibirPromociones;
    }

    // Método para calcular el descuento basado en la frecuencia de
visitas
    public double calcularDescuento() {
        if (visitas >= 3 && visitas < 5) {
            return 0.10; // 10% de descuento
        } else if (visitas >= 5) {
            return 0.20; // 20% de descuento
        }
        return 0.0; // Sin descuento
    }

    // Método para verificar si el cliente cumple con los requisitos de
estatura
    public boolean cumpleRequisitosEstatura(double alturaMinima) {
        return this.estatura >= alturaMinima;
    }
}

```

### ➤ Atributos

✓ **nombre:** Almacena el nombre del cliente.

**cedula:** Identificación del cliente, probablemente un número de documento.

**telefono:** Número de contacto del cliente.

**correo:** Dirección de correo electrónico del cliente.

**estatura:** Altura del cliente, en metros o centímetros.

**edad:** Edad del cliente, en años.

**contactoFamiliar:** Información de contacto de un familiar, utilizada solo si el cliente es menor de edad.

**visitas:** Contador que registra cuántas veces ha visitado el parque el cliente.

**esFrecuente:** Booleano que indica si el cliente es considerado frecuente (5 o más visitas).

**puedeRecibirPromociones:** Booleano que indica si el cliente puede recibir promociones (solo mayores de edad).

### ➤ Constructor

El constructor inicializa todos los atributos del cliente. Además, establece el contacto familiar solo si el cliente es menor de edad, determina si el cliente es frecuente basado en el número de visitas y verifica si puede recibir promociones según su edad.

- **Getters y Setters:** Métodos para acceder y modificar los atributos de la clase. Incluyen validaciones específicas para **edad** y **visitas** que actualizan automáticamente los atributos **puedeRecibirPromociones** y **esFrecuente**.
- **calcularDescuento():** Este método calcula un descuento basado en el número de visitas del cliente. Si el cliente ha visitado entre 3 y 4 veces, recibe un 10% de descuento; si ha visitado 5 o más veces, recibe un 20% de descuento. Si ha visitado menos de 3 veces, no recibe descuento.
- **cumpleRequisitosEstatura(double alturaMinima):** Verifica si la estatura del cliente cumple con un requisito mínimo especificado.

2. **Empleado:** Representa a los empleados del parque, con atributos como nombre, cédula, teléfono, rol y horario laboral.

```
package com.salitreparquemagicoapp.modelo;

public class Empleado {
    private String nombre;
    private String cedula;
    private String telefono;
    private String correo;
    private String cargo; // Tipos de empleados: administrativo,
logística, publicidad, operadores, mantenimiento
    private String horarioLaboral;

    // Constructor
    public Empleado(String nombre, String cedula, String telefono, String
```



```
correo, String cargo, String horarioLaboral) {
    this.nombre = nombre;
    this.cedula = cedula;
    this.telefono = telefono;
    this.correo = correo;
    this.cargo = cargo;
    this.horarioLaboral = horarioLaboral;
}

// Getters y Setters
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getCedula() {
    return cedula;
}

public void setCedula(String cedula) {
    this.cedula = cedula;
}

public String getTelefono() {
    return telefono;
}

public void setTelefono(String telefono) {
    this.telefono = telefono;
}

public String getCorreo() {
    return correo;
}

public void setCorreo(String correo) {
    this.correo = correo;
}

public String getCargo() {
    return cargo;
}

public void setCargo(String cargo) {
    this.cargo = cargo;
}

public String getHorarioLaboral() {
    return horarioLaboral;
}
```

```

public void setHorarioLaboral(String horarioLaboral) {
    this.horarioLaboral = horarioLaboral;
}

// Método específico para operadores
public void verificarIngreso(Cliente cliente, Atraccion atraccion) {
    if
(!cliente.cumpleRequisitosEstatura(atraccion.getAlturaMinima())) {
        System.out.println("Alerta: " + cliente.getNombre() + " no
cumple con la estatura mínima para la atracción " +
atraccion.getNombre());
    } else if (!atraccion.isHabilitada()) {
        System.out.println("Alerta: La atracción " +
atraccion.getNombre() + " no está habilitada.");
    } else {
        System.out.println("Ingreso permitido a " +
cliente.getNombre() + " para la atracción " + atraccion.getNombre());
    }
}

// Método para que el empleado de mantenimiento reporte el estado de
una máquina
public void reportarEstadoMaquina(Estacion estacion, boolean estado)
{
    estacion.setMaquinaDisponible(estado);
    System.out.println("Estado de la máquina en " +
estacion.getNombre() + " reportado como " + (estado ? "disponible" : "no
disponible"));
}

// Método para que el empleado administrativo habilite o deshabilite
una atracción
public void habilitarAtraccion(Atraccion atraccion, boolean
habilitada) {
    atraccion.setHabilitada(habilitada);
    System.out.println("La atracción " + atraccion.getNombre() + " ha
sido " + (habilitada ? "habilitada" : "deshabilitada"));
}
}

```

## Descripción de la Clase Empleado

### ➤ Atributos:

**nombre:** Nombre del empleado.

**cedula:** Identificación del empleado.

**telefono:** Número de teléfono del empleado.

**correo:** Correo electrónico del empleado.

**cargo:** Tipo de empleado (administrativo, logística, publicidad, operadores, mantenimiento).

**horarioLaboral:** Horario laboral del empleado.

- **Constructor:** Inicializa todos los atributos.
- **Getters y Setters:** Métodos para acceder y modificar los atributos de la clase.
- **Método verificarIngreso:** Este método es específico para los empleados de tipo "operador". Permite verificar la estatura y el estado de habilitación de una atracción para decidir si se le permite el ingreso a un cliente.
- **Método reportarEstadoMaquina:** Permite a los empleados de mantenimiento reportar el estado de una máquina en una estación (disponible o no disponible).
- **Método habilitarAtraccion:** Permite a los empleados administrativos habilitar o deshabilitar una atracción.

Esta clase es fundamental para gestionar la información de los empleados en el sistema del parque de diversiones y para aplicar las reglas de negocio relacionadas con sus responsabilidades.

3. **Atraccion:** Representa las atracciones del parque, con atributos como nombre, descripción, clasificación, estatura mínima, disponibilidad y visitas.

```
package com.salitreparquemagicoapp.modelo;

public class Atraccion {
    private String nombre;
    private String descripcion;
    private double costo;
    private double alturaMinima;
    private int edadMinima;
    private String clasificacion; // Clasificación de la atracción
    private String condicionesUso; // Condiciones de uso
    private boolean habilitada; // Estado de habilitación
    private int contadorVisitas; // Contador de visitas a la atracción

    // Constructor
    public Atraccion(String nombre, String descripcion, double costo,
double alturaMinima, int edadMinima, String clasificacion, String
condicionesUso) {
        this.nombre = nombre;
    }
}
```

```

        this.descripcion = descripcion;
        this.costo = costo;
        this.alturaMinima = alturaMinima;
        this.edadMinima = edadMinima;
        this.clasificacion = clasificacion;
        this.condicionesUso = condicionesUso;
        this.habilitada = true; // Por defecto, la atracción está
habilitada
        this.contadorVisitas = 0; // Inicialmente, no tiene visitas
    }

    // Getters y Setters
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public double getCosto() {
        return costo;
    }

    public void setCosto(double costo) {
        this.costo = costo;
    }

    public double getAlturaMinima() {
        return alturaMinima;
    }

    public void setAlturaMinima(double alturaMinima) {
        this.alturaMinima = alturaMinima;
    }

    public int getEdadMinima() {
        return edadMinima;
    }

    public void setEdadMinima(int edadMinima) {
        this.edadMinima = edadMinima;
    }

    public String getClasificacion() {
        return clasificacion;
    }

```

```

    }

    public void setClasificacion(String clasificacion) {
        this.clasificacion = clasificacion;
    }

    public String getCondicionesUso() {
        return condicionesUso;
    }

    public void setCondicionesUso(String condicionesUso) {
        this.condicionesUso = condicionesUso;
    }

    public boolean isHabilitada() {
        return habilitada;
    }

    public void setHabilitada(boolean habilitada) {
        this.habilitada = habilitada;
    }

    public int getContadorVisitas() {
        return contadorVisitas;
    }

    // Método para incrementar el contador de visitas
    public void incrementarVisitas() {
        this.contadorVisitas++;
    }
}

```

## Descripción de la Clase Atraccion

### ➤ Atributos:

**nombre:** Nombre de la atracción.

**descripcion:** Descripción de la atracción.

**costo:** Costo de la atracción.

**alturaMinima:** Altura mínima requerida para acceder a la atracción.

**edadMinima:** Edad mínima requerida para acceder a la atracción.

**clasificacion:** Clasificación de la atracción (por ejemplo, "aventura", "infantil", etc.).

**condicionesUso:** Condiciones de uso de la atracción.

**habilitada:** Estado de habilitación de la atracción (si está disponible para los clientes).

**contadorVisitas:** Contador de visitas a la atracción.

- **Constructor:** Inicializa todos los atributos y establece que la atracción está habilitada por defecto.
- **Getters y Setters:** Métodos para acceder y modificar los atributos de la clase.
- **Método incrementarVisitas:** Incrementa el contador de visitas a la atracción cada vez que un cliente la utiliza.

Esta clase es fundamental para gestionar la información de las atracciones en el sistema del parque de diversiones y para aplicar las reglas de negocio relacionadas con su uso y disponibilidad.

#### 4. Tiquete:

```
package com.salitreparquemagicoapp.modelo;

import java.util.Date;

public class Tiquete {
    private String id; // Identificador único del tiquete
    private double precio; // Costo del tiquete
    private Date fechaCompra; // Fecha de compra del tiquete
    private boolean esFrecuente; // Indica si el tiquete es para un
    cliente frecuente
    private Date fechaValidez; // Fecha de validez del tiquete
    private String cedulaCliente; // Cédula del cliente que compra el
    tiquete
    private int visitasCliente; // Número de visitas del cliente

    // Constructor
    public Tiquete(String id, double precio, Date fechaCompra, String
    cedulaCliente, int visitasCliente) {
        this.id = id;
        this.precio = precio;
        this.fechaCompra = fechaCompra;
        this.cedulaCliente = cedulaCliente;
        this.visitasCliente = visitasCliente;
        this.fechaValidez = calcularFechaValidez(); // Establecer la
    fecha de validez
        this.esFrecuente = visitasCliente >= 5; // Determinar si el
    cliente es frecuente
    }

    // Getters y Setters
    public String getId() {
        return id;
    }
}
```

```
public void setId(String id) {
    this.id = id;
}

public double getPrecio() {
    return precio;
}

public void setPrecio(double precio) {
    this.precio = precio;
}

public Date getFechaCompra() {
    return fechaCompra;
}

public void setFechaCompra(Date fechaCompra) {
    this.fechaCompra = fechaCompra;
}

public boolean isEsFrecuente() {
    return esFrecuente;
}

public void setEsFrecuente(boolean esFrecuente) {
    this.esFrecuente = esFrecuente;
}

public Date getFechaValidez() {
    return fechaValidez;
}

public void setFechaValidez(Date fechaValidez) {
    this.fechaValidez = fechaValidez;
}

public String getCedulaCliente() {
    return cedulaCliente;
}

public void setCedulaCliente(String cedulaCliente) {
    this.cedulaCliente = cedulaCliente;
}

public int getVisitasCliente() {
    return visitasCliente;
}

public void setVisitasCliente(int visitasCliente) {
    this.visitasCliente = visitasCliente;
    this.esFrecuente = visitasCliente >= 5; // Actualizar si el
cliente es frecuente
}
```

```
// Método para calcular la fecha de validez (por ejemplo, 30 días después de la compra)
private Date calcularFechaValidez() {
    long milisegundosEnUnDia = 1000 * 60 * 60 * 24;
    return new Date(fechaCompra.getTime() + (30 * milisegundosEnUnDia)); // 30 días de validez
}

// Método para calcular el precio con descuento basado en la frecuencia de visitas
public double calcularPrecioConDescuento() {
    double descuento = 0.0;
    if (visitasCliente >= 3 && visitasCliente < 5) {
        descuento = 0.10; // 10% de descuento
    } else if (visitasCliente >= 5) {
        descuento = 0.20; // 20% de descuento
    }
    return precio * (1 - descuento); // Aplicar el descuento al precio
}

// Método para verificar si el ticket es válido
public boolean esValido() {
    Date hoy = new Date();
    return hoy.before(fechaValidez);
}
}
```

## Descripción de la Clase Tiquete

### ➤ Atributos:

**id:** Identificador único del tiquete, que puede ser un número o un código alfanumérico.

**costo:** Costo del tiquete, que puede variar según el tipo de tiquete (general, familiar, etc.).

**fechaCompra:** Fecha en la que se compró el tiquete.

**esFrecuente:** Indica si el tiquete es para un cliente frecuente, lo que puede influir en el costo o en los beneficios asociados.

**fechaValidez:** Fecha hasta la cual el tiquete es válido para su uso.

➤ **Constructor:** Inicializa todos los atributos del tiquete.

➤ **Getters y Setters:** Métodos para acceder y modificar los atributos de la clase.



- **Método esValido:** Verifica si el ticket es válido comparando la fecha actual con la fecha de validez.

Esta clase es fundamental para gestionar la información de los tickets en el sistema del parque de diversiones, permitiendo a los empleados y al sistema en general verificar la validez de los tickets y gestionar el acceso de los clientes al parque.

5. **Estacion:** Representa las estaciones donde se venden los tickets.

```
package com.salitreparquemagicoapp.modelo;

public class Estacion {
    private String nombre;
    private String ubicacion;
    private int capacidad;
    private boolean habilitada; // Estado de habilitación
    private boolean maquinaDisponible; // Estado de la máquina

    // Constructor
    public Estacion(String nombre, String ubicacion, int capacidad) {
        this.nombre = nombre;
        this.ubicacion = ubicacion;
        this.capacidad = capacidad;
        this.habilitada = true; // Por defecto, la estación está
        this.maquinaDisponible = true; // Por defecto, la máquina está
    }

    // Getters y Setters
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getUbicacion() {
        return ubicacion;
    }

    public void setUbicacion(String ubicacion) {
        this.ubicacion = ubicacion;
    }

    public int getCapacidad() {
        return capacidad;
    }
}
```

```
public void setCapacidad(int capacidad) {
    this.capacidad = capacidad;
}

public boolean isHabilitada() {
    return habilitada;
}

public void setHabilitada(boolean habilitada) {
    this.habilitada = habilitada;
}

public boolean isMaquinaDisponible() {
    return maquinaDisponible;
}

public void setMaquinaDisponible(boolean maquinaDisponible) {
    this.maquinaDisponible = maquinaDisponible;
}
}
```

### Descripción de la Clase Estacion:

#### ➤ Atributos:

**nombre:** Nombre de la estación (por ejemplo, "Estación de Juegos", "Estación de Comida", etc.).

**maquinaDisponible:** Estado de la máquina en la estación, que indica si está disponible para su uso o no.

➤ **Constructor:** Inicializa el nombre de la estación y establece que la máquina está disponible por defecto.

➤ **Getters y Setters:** Métodos para acceder y modificar los atributos de la clase.

Esta clase es fundamental para gestionar la información de las estaciones en el sistema del parque de diversiones, permitiendo a los empleados y al sistema en general verificar el estado de las máquinas y realizar las operaciones necesarias en función de su disponibilidad.

```
package com.salitreparquemagicoapp.modelo;

public class Estacion {
    private String nombre;
    private String descripcion;
```

```

public Estacion(String nombre, String descripcion) {
    this.nombre = nombre;
    this.descripcion = descripcion;
}

// Getters y Setters
public String getNombre() { return nombre; }
public String getDescripcion() { return descripcion; }

@Override
public String toString() {
    return "Estacion{" +
        "nombre='" + nombre + '\'' +
        ", descripcion='" + descripcion + '\'' +
        '}';
}
}

```

**Paquete Vista:** Se encarga del diseño y presentación de los datos.

#### VistaConsola:

- ✓ **Interacción con el Usuario:** La clase VistaConsola permite al usuario interactuar con el sistema a través de un menú en la consola, donde puede elegir entre gestionar clientes, empleados, atracciones, tiquetes y estaciones.
- ✓ **Gestión de Clientes:** Métodos para agregar, listar, actualizar y eliminar clientes, incluyendo la validación de edad para el contacto familiar.
- ✓ **Gestión de Empleados:** Métodos para agregar, listar, actualizar y eliminar empleados.
- ✓ **Gestión de Atracciones:** Métodos para agregar, listar, actualizar y eliminar atracciones, permitiendo al usuario ingresar detalles como nombre, descripción y altura mínima.
- ✓ **Gestión de Tiquetes:** Métodos para agregar, listar y eliminar tiquetes, donde se solicita el código y el precio.
- ✓ **Gestión de Estaciones:** Métodos para agregar, listar, actualizar y eliminar estaciones, permitiendo al usuario ingresar el nombre y la descripción de cada estación.

Esta implementación proporciona una interfaz de usuario sencilla y efectiva para gestionar las diferentes entidades del parque, facilitando la interacción y el manejo de datos, Su código es el siguiente.

```
package com.salitreparquemagicoapp.vista;

import com.salitreparquemagicoapp.controlador.ControladorParque;
import com.salitreparquemagicoapp.modelo.Atraccion;
import com.salitreparquemagicoapp.modelo.Cliente;
import com.salitreparquemagicoapp.modelo.Empleado;
import com.salitreparquemagicoapp.modelo.Estacion;
import com.salitreparquemagicoapp.modelo.Tiquete;

import java.sql.Date;
import java.util.List;
import java.util.Scanner;

public class VistaConsola {
    private ControladorParque controlador;
    private Scanner scanner;

    public VistaConsola () {
        controlador = new ControladorParque();
        scanner = new Scanner(System.in);
    }

    public void mostrarMenu () {
        while (true) {
            System.out.println("\n--- Menú Parque Salitre Magico ---");
            System.out.println("1. Gestión de Clientes");
            System.out.println("2. Gestión de Empleados");
            System.out.println("3. Gestión de Estaciones");
            System.out.println("4. Gestión de Atracciones");
            System.out.println("5. Gestión de Tiquetes");
            System.out.println("0. Salir");
            System.out.print("Seleccione una opción: ");

            int opcion = scanner.nextInt();
            scanner.nextLine(); // Limpiar el buffer

            switch (opcion) {
                case 1:
                    gestionarClientes();
                    break;
                case 2:
                    gestionarEmpleados();
                    break;
                case 3:
                    gestionarEstaciones();
                    break;
                case 4:
                    gestionarAtracciones();
                    break;
                case 5:
                    gestionarTiquetes();
                    break;
                case 0:
                    System.out.println("Saliendo...");
            }
        }
    }
}
```

```

        return;
    default:
        System.out.println("Opción no válida. Intente de
nuevo.");
    }
}

// Métodos para gestionar Clientes
private void gestionarClientes () {
    while (true) {
        System.out.println("\n--- Gestión de Clientes ---");
        System.out.println("1. Agregar Cliente");
        System.out.println("2. Listar Clientes");
        System.out.println("3. Actualizar Cliente");
        System.out.println("4. Eliminar Cliente");
        System.out.println("0. Volver al menú principal");
        System.out.print("Seleccione una opción: ");

        int opcion = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer

        switch (opcion) {
            case 1:
                agregarCliente();
                break;
            case 2:
                listarClientes();
                break;
            case 3:
                actualizarCliente();
                break;
            case 4:
                eliminarCliente();
                break;
            case 0:
                return;
            default:
                System.out.println("Opción no válida. Intente de
nuevo.");
        }
    }
}

private void agregarCliente () {
    System.out.print("Ingrese el nombre del cliente: ");
    String nombre = scanner.nextLine();
    System.out.print("Ingrese la cédula: ");
    String cedula = scanner.nextLine();
    System.out.print("Ingrese el teléfono: ");
    String telefono = scanner.nextLine();
    System.out.print("Ingrese el correo: ");
    String correo = scanner.nextLine();
    System.out.print("Ingrese la estatura: ");

```

```

double estatura = scanner.nextDouble();
System.out.print("Ingrese la edad: ");
int edad = scanner.nextInt();
scanner.nextLine(); // Limpiar el buffer

String contactoFamiliar = null;
if (edad < 18) {
    System.out.print("Ingrese el contacto familiar: ");
    contactoFamiliar = scanner.nextLine();
}

System.out.print("Ingrese el número de visitas: ");
int visitas = scanner.nextInt();

Cliente cliente = new Cliente(nombre, cedula, telefono, correo,
estatura, edad, contactoFamiliar, visitas);
controlador.registrarCliente(cliente);
System.out.println("Cliente registrado exitosamente.");
}

private void listarClientes () {
    List<Cliente> clientes = controlador.listarClientes();
    System.out.println("Lista de Clientes:");
    for (Cliente cliente : clientes) {
        System.out.println("Nombre: " + cliente.getNombre() + ",
Cédula: " + cliente.getCedula() +
        ", Puede recibir promociones: " +
(cliente.setPuedeRecibirPromociones() ? "Sí" : "No"));
    }
}

private void actualizarCliente () {
    System.out.print("Ingrese la cédula del cliente a actualizar: ");
    String cedula = scanner.nextLine();
    Cliente cliente = controlador.listarClientes().stream()
        .filter(c -> c.getCedula().equals(cedula))
        .findFirst().orElse(null);
    if (cliente != null) {
        System.out.print("Ingrese el nuevo nombre del cliente: ");
        String nombre = scanner.nextLine();
        System.out.print("Ingrese el nuevo teléfono: ");
        String telefono = scanner.nextLine();
        System.out.print("Ingrese el nuevo correo: ");
        String correo = scanner.nextLine();
        System.out.print("Ingrese la nueva estatura: ");
        double estatura = scanner.nextDouble();
        System.out.print("Ingrese la nueva edad: ");
        int edad = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer

        String contactoFamiliar = null;
        if (edad < 18) {
            System.out.print("Ingrese el nuevo contacto familiar: ");
            contactoFamiliar = scanner.nextLine();

```

```

    }

    System.out.print("Ingrese el nuevo número de visitas: ");
    int visitas = scanner.nextInt();

    cliente.setNombre(nombre);
    cliente.setTelefono(telefono);
    cliente.setCorreo(correo);
    cliente.setEstatura(estatura);
    cliente.setEdad(edad);
    cliente.setContactoFamiliar(contactoFamiliar);
    cliente.setVisitas(visitas);
    controlador.actualizarCliente(cliente);
    System.out.println("Cliente actualizado exitosamente.");
} else {
    System.out.println("Cliente no encontrado.");
}
}

private void eliminarCliente () {
    System.out.print("Ingrese la cédula del cliente a eliminar: ");
    String cedula = scanner.nextLine();
    boolean eliminado = controlador.eliminarCliente(cedula);
    if (eliminado) {
        System.out.println("Cliente eliminado exitosamente.");
    } else {
        System.out.println("Cliente no encontrado.");
    }
}

// Métodos para gestionar Empleados
private void gestionarEmpleados () {
    while (true) {
        System.out.println("\n--- Gestión de Empleados ---");
        System.out.println("1. Agregar Empleado");
        System.out.println("2. Listar Empleados");
        System.out.println("3. Actualizar Empleado");
        System.out.println("4. Eliminar Empleado");
        System.out.println("0. Volver al menú principal");
        System.out.print("Seleccione una opción: ");

        int opcion = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer

        switch (opcion) {
            case 1:
                agregarEmpleado();
                break;
            case 2:
                listarEmpleados();
                break;
            case 3:
                actualizarEmpleado();
                break;
        }
    }
}

```

```

        case 4:
            eliminarEmpleado();
            break;
        case 0:
            return;
        default:
            System.out.println("Opción no válida. Intente de nuevo.");
    }
}

private void agregarEmpleado () {
    System.out.print("Ingrese el nombre del empleado: ");
    String nombre = scanner.nextLine();
    System.out.print("Ingrese la cédula: ");
    String cedula = scanner.nextLine();
    System.out.print("Ingrese el teléfono: ");
    String telefono = scanner.nextLine();
    System.out.print("Ingrese el correo: ");
    String correo = scanner.nextLine();
    System.out.print("Ingrese el cargo (administrativo, logística, publicidad, operadores, mantenimiento): ");
    String cargo = scanner.nextLine();
    System.out.print("Ingrese el horario laboral: ");
    String horarioLaboral = scanner.nextLine();

    Empleado empleado = new Empleado(nombre, cedula, telefono, correo, cargo, horarioLaboral);
    controlador.registrarEmpleado(empleado);
    System.out.println("Empleado registrado exitosamente.");
}

private void listarEmpleados () {
    List<Empleado> empleados = controlador.listarEmpleados();
    System.out.println("Lista de Empleados:");
    System.out.printf("%-20s %-15s %-15s %-40s %-30s %-20s\n",
"Nombre", "Cédula", "Teléfono", "Correo", "Cargo", "Horario Laboral");
    System.out.println("-----");
    System.out.println("-----");

    for (Empleado empleado : empleados) {
        System.out.printf("%-20s %-15s %-15s %-40s %-30s %-20s\n",
            empleado.getNombre(),
            empleado.getCedula(),
            empleado.getTelefono(),
            empleado.getCorreo(),
            empleado.getCargo(),
            empleado.getHorarioLaboral());
    }
}

private void actualizarEmpleado () {
    System.out.print("Ingrese la cédula del empleado a actualizar: ");
}

```



```

String cedula = scanner.nextLine();
Empleado empleado = controlador.listarEmpleados().stream()
    .filter(e -> e.getCedula().equals(cedula))
    .findFirst().orElse(null);
if (empleado != null) {
    System.out.print("Ingrese el nuevo nombre del empleado
(actual: " + empleado.getNombre() + "): ");
    String nombre = scanner.nextLine();
    System.out.print("Ingrese el nuevo teléfono (actual: " +
empleado.getTelefono() + "): ");
    String telefono = scanner.nextLine();
    System.out.print("Ingrese el nuevo correo (actual: " +
empleado.getCorreo() + "): ");
    String correo = scanner.nextLine();
    System.out.print("Ingrese el nuevo cargo (actual: " +
empleado.getCargo() + "): ");
    String cargo = scanner.nextLine();
    System.out.print("Ingrese el nuevo horario laboral (actual: "
+ empleado.getHorarioLaboral() + "): ");
    String horarioLaboral = scanner.nextLine();

    empleado.setNombre(nombre);
    empleado.setTelefono(telefono);
    empleado.setCorreo(correo);
    empleado.setCargo(cargo);
    empleado.setHorarioLaboral(horarioLaboral);
    controlador.actualizarEmpleado(empleado);
    System.out.println("Empleado actualizado exitosamente.");
} else {
    System.out.println("Empleado no encontrado.");
}

private void eliminarEmpleado () {
    System.out.print("Ingrese la cédula del empleado a eliminar: ");
    String cedula = scanner.nextLine();
    boolean eliminado = controlador.eliminarEmpleado(cedula);
    if (eliminado) {
        System.out.println("Empleado eliminado exitosamente.");
    } else {
        System.out.println("Empleado no encontrado.");
    }
}

// Métodos para gestionar Estaciones
private void gestionarEstaciones () {
    while (true) {
        System.out.println("\n--- Gestión de Estaciones ---");
        System.out.println("1. Agregar Estación");
        System.out.println("2. Listar Estaciones");
        System.out.println("3. Actualizar Estación");
        System.out.println("4. Eliminar Estación");
        System.out.println("0. Volver al menú principal");
        System.out.print("Seleccione una opción: ");
    }
}

```

```

        int opcion = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer

        switch (opcion) {
            case 1:
                agregarEstacion();
                break;
            case 2:
                listarEstaciones();
                break;
            case 3:
                actualizarEstacion();
                break;
            case 4:
                eliminarEstacion();
                break;
            case 0:
                return;
            default:
                System.out.println("Opción no válida. Intente de
nuevo.");
        }
    }

    private void agregarEstacion () {
        System.out.print("Ingrese el nombre de la estación: ");
        String nombre = scanner.nextLine();
        System.out.print("Ingrese la ubicación de la estación: ");
        String ubicacion = scanner.nextLine();
        System.out.print("Ingrese la capacidad de la estación: ");
        int capacidad = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer
        System.out.print("¿Está habilitada? (true/false): ");
        boolean habilitada = scanner.nextBoolean();
        scanner.nextLine(); // Limpiar el buffer
        System.out.print("¿Está la máquina disponible? (true/false): ");
        boolean maquinaDisponible = scanner.nextBoolean();
        scanner.nextLine(); // Limpiar el buffer

        Estacion estacion = new Estacion(nombre, ubicacion, capacidad);
        estacion.setHabilitada(habilitada);
        estacion.setMaquinaDisponible(maquinaDisponible); // Cambiado a
setMaquinaDisponible
        controlador.registrarEstacion(estacion);
        System.out.println("Estación registrada exitosamente.");
    }

    private void listarEstaciones () {
        List<Estacion> estaciones = controlador.listarEstaciones();
        System.out.println("Lista de Estaciones:");
        for (Estacion estacion : estaciones) {
            System.out.println("Nombre: " + estacion.getNombre() +

```

```

        ", Ubicación: " + estacion.getUbicacion() +
        ", Capacidad: " + estacion.getCapacidad() +
        ", Habilitada: " + estacion.isHabilitada() +
        ", Máquina disponible: " +
estacion.isMaquinaDisponible()); // Mostrar estado de la máquina
    }
}

private void actualizarEstacion () {
    System.out.print("Ingrese el nombre de la estación a actualizar:
");
    String nombre = scanner.nextLine();
    Estacion estacion = controlador.listarEstaciones().stream()
        .filter(e -> e.getNombre().equals(nombre))
        .findFirst().orElse(null);
    if (estacion != null) {
        System.out.print("Ingrese la nueva ubicación: ");
        String ubicacion = scanner.nextLine();
        System.out.print("Ingrese la nueva capacidad: ");
        int capacidad = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer
        System.out.print("¿Está habilitada? (true/false): ");
        boolean habilitada = scanner.nextBoolean();
        scanner.nextLine(); // Limpiar el buffer
        System.out.print("¿Está la máquina disponible? (true/false):
");
        boolean maquinaDisponible = scanner.nextBoolean();
        scanner.nextLine(); // Limpiar el buffer

        estacion.setUbicacion(ubicacion);
        estacion.setCapacidad(capacidad);
        estacion.setHabilitada(habilitada);
        estacion.setMaquinaDisponible(maquinaDisponible); // Cambiado
a setMaquinaDisponible
        controlador.actualizarEstacion(estacion);
        System.out.println("Estación actualizada exitosamente.");
    } else {
        System.out.println("Estación no encontrada.");
    }
}

private void eliminarEstacion () {
    System.out.print("Ingrese el nombre de la estación a eliminar:
");
    String nombre = scanner.nextLine();
    boolean eliminado = controlador.eliminarEstacion(nombre);
    if (eliminado) {
        System.out.println("Estación eliminada exitosamente.");
    } else {
        System.out.println("Estación no encontrada.");
    }
}

// Métodos para gestionar Atracciones

```

```
private void gestionarAtracciones () {
    while (true) {
        System.out.println("\n--- Gestión de Atracciones ---");
        System.out.println("1. Agregar Atracción");
        System.out.println("2. Listar Atracciones");
        System.out.println("3. Actualizar Atracción");
        System.out.println("4. Eliminar Atracción");
        System.out.println("0. Volver al menú principal");
        System.out.print("Seleccione una opción: ");

        int opcion = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer

        switch (opcion) {
            case 1:
                agregarAtraccion();
                break;
            case 2:
                listarAtracciones();
                break;
            case 3:
                actualizarAtraccion();
                break;
            case 4:
                eliminarAtraccion();
                break;
            case 0:
                return;
            default:
                System.out.println("Opción no válida. Intente de nuevo.");
        }
    }
}

private void agregarAtraccion () {
    System.out.print("Ingrese el nombre de la atracción: ");
    String nombre = scanner.nextLine();
    System.out.print("Ingrese la descripción: ");
    String descripcion = scanner.nextLine();
    System.out.print("Ingrese el costo: ");
    double costo = scanner.nextDouble();
    System.out.print("Ingrese la altura mínima: ");
    double alturaMinima = scanner.nextDouble();
    System.out.print("Ingrese la edad mínima: ");
    int edadMinima = scanner.nextInt();
    scanner.nextLine(); // Limpiar el buffer
    System.out.print("Ingrese la clasificación: ");
    String clasificacion = scanner.nextLine();
    System.out.print("Ingrese las condiciones de uso: ");
    String condicionesUso = scanner.nextLine();

    Atraccion atraccion = new Atraccion(nombre, descripcion, costo,
    alturaMinima, edadMinima, clasificacion, condicionesUso);
}
```

```

        controlador.registrarAtraccion(atraccion);
        System.out.println("Atracción registrada exitosamente.");
    }

    private void listarAtracciones () {
        List<Atraccion> atracciones = controlador.listarAtracciones();
        System.out.println("Lista de Atracciones:");
        for (Atraccion atraccion : atracciones) {
            System.out.println("Nombre: " + atraccion.getNombre() +
                ", Descripción: " + atraccion.getDescripcion() +
                ", Costo: " + atraccion.getCosto() +
                ", Altura mínima: " + atraccion.getAlturaMinima() +
                ", Edad mínima: " + atraccion.getEdadMinima() +
                ", Clasificación: " + atraccion.getClasificacion() +
                ", Condiciones de uso: " +
atraccion.getCondicionesUso() +
                ", Habilitada: " + atraccion.isHabilitada());
        }
    }

    private void actualizarAtraccion () {
        System.out.print("Ingrese el nombre de la atracción a actualizar:");

        String nombre = scanner.nextLine();
        Atraccion atraccion = controlador.listarAtracciones().stream()
            .filter(a -> a.getNombre().equals(nombre))
            .findFirst().orElse(null);
        if (atraccion != null) {
            System.out.print("Ingrese la nueva descripción: ");
            String descripcion = scanner.nextLine();
            System.out.print("Ingrese el nuevo costo: ");
            double costo = scanner.nextDouble();
            System.out.print("Ingrese la nueva altura mínima: ");
            double alturaMinima = scanner.nextDouble();
            System.out.print("Ingrese la nueva edad mínima: ");
            int edadMinima = scanner.nextInt();
            scanner.nextLine(); // Limpiar el buffer
            System.out.print("Ingrese la nueva clasificación: ");
            String clasificacion = scanner.nextLine();
            System.out.print("Ingrese las nuevas condiciones de uso: ");
            String condicionesUso = scanner.nextLine();

            atraccion.setDescripcion(descripcion);
            atraccion.setCosto(costo);
            atraccion.setAlturaMinima(alturaMinima);
            atraccion.setEdadMinima(edadMinima);
            atraccion.setClasificacion(clasificacion);
            atraccion.setCondicionesUso(condicionesUso);
            controlador.actualizarAtraccion(atraccion);
            System.out.println("Atracción actualizada exitosamente.");
        } else {
            System.out.println("Atracción no encontrada.");
        }
    }
}

```

```

private void eliminarAtraccion () {
    System.out.print("Ingrese el nombre de la atracción a eliminar:
");
    String nombre = scanner.nextLine();
    boolean eliminado = controlador.eliminarAtraccion(nombre);
    if (eliminado) {
        System.out.println("Atracción eliminada exitosamente.");
    } else {
        System.out.println("Atracción no encontrada.");
    }
}

// Métodos para gestionar Tiquetes
private void gestionarTiquetes () {
    while (true) {
        System.out.println("\n--- Gestión de Tiquetes ---");
        System.out.println("1. Agregar Tiquete");
        System.out.println("2. Listar Tiquetes");
        System.out.println("3. Actualizar Tiquete");
        System.out.println("4. Eliminar Tiquete");
        System.out.println("0. Volver al menú principal");
        System.out.print("Seleccione una opción: ");

        int opcion = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer

        switch (opcion) {
            case 1:
                agregarTiquete();
                break;
            case 2:
                listarTiquetes();
                break;
            case 3:
                actualizarTiquete();
                break;
            case 4:
                eliminarTiquete();
                break;
            case 0:
                return;
            default:
                System.out.println("Opción no válida. Intente de
nuevo.");
        }
    }
}

private void agregarTiquete () {
    System.out.print("Ingrese el ID del tiquete: ");
    String id = scanner.nextLine();
    System.out.print("Ingrese el precio: ");
    double precio = scanner.nextDouble();

```

```

scanner.nextLine(); // Limpiar el buffer
System.out.print("Ingrese la fecha de compra (YYYY-MM-DD): ");
String fechaCompraStr = scanner.nextLine();
Date fechaCompra = Date.valueOf(fechaCompraStr);

// Solicitar la cédula del cliente
System.out.print("Ingrese la cédula del cliente: ");
String cedulaCliente = scanner.nextLine();

// Solicitar el número de visitas del cliente
System.out.print("Ingrese el número de visitas del cliente: ");
int visitasCliente = scanner.nextInt();
scanner.nextLine(); // Limpiar el buffer

// Crear el ticket
Ticket ticket = new Ticket(id, precio, fechaCompra,
cedulaCliente, visitasCliente);
controlador.registrarTicket(ticket);
System.out.println("Ticket registrado exitosamente.");
}

private void listarTickets () {
    List<Ticket> tickets = controlador.listarTickets();
    System.out.println("Lista de Tickets:");
    for (Ticket ticket : tickets) {
        System.out.println("ID: " + ticket.getId() +
            ", Precio: " + ticket.calcularPrecioConDescuento() +
// Mostrar precio con descuento
            ", Fecha de compra: " + ticket.getFechaCompra() +
            ", Cédula del cliente: " + ticket.getCedulaCliente()
+ // Mostrar cédula del cliente
            ", Visitas del cliente: " +
ticket.getVisitasCliente() + // Mostrar visitas del cliente
            ", Fecha de validez: " + ticket.getFechaValidez());
    }
}

private void actualizarTicket () {
    System.out.print("Ingrese el ID del ticket a actualizar: ");
    String id = scanner.nextLine();
    Ticket ticket = controlador.listarTickets().stream()
        .filter(t -> t.getId().equals(id))
        .findFirst().orElse(null);
    if (ticket != null) {
        System.out.print("Ingrese el nuevo precio: ");
        double precio = scanner.nextDouble();
        scanner.nextLine(); // Limpiar el buffer
        System.out.print("Ingrese la nueva fecha de compra (YYYY-MM-
DD): ");
        String fechaCompraStr = scanner.nextLine();
        Date fechaCompra = Date.valueOf(fechaCompraStr);

        // Solicitar la nueva cédula del cliente
        System.out.print("Ingrese la nueva cédula del cliente: ");

```

```

        String cedulaCliente = scanner.nextLine();

        // Solicitar el nuevo número de visitas del cliente
        System.out.print("Ingrese el nuevo número de visitas del
cliente: ");
        int visitasCliente = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer
        System.out.print("Ingrese la nueva fecha de validez (YYYY-MM-
DD): ");

        String fechaValidezStr = scanner.nextLine();
        Date fechaValidez = Date.valueOf(fechaValidezStr);

        tiquete.setPrecio(precio);
        tiquete.setFechaCompra(fechaCompra);
        tiquete.setCedulaCliente(cedulaCliente); // Actualizar cédula
del cliente
        tiquete.setVisitasCliente(visitasCliente); // Actualizar
número de visitas del cliente
        tiquete.setFechaValidez(fechaValidez);
        controlador.actualizarTiquete(tiquete);
        System.out.println("Tiquete actualizado exitosamente.");
    } else {
        System.out.println("Tiquete no encontrado.");
    }
}

private void eliminarTiquete () {
    System.out.print("Ingrese el ID del tiquete a eliminar: ");
    String id = scanner.nextLine();
    boolean eliminado = controlador.eliminarTiquete(id);
    if (eliminado) {
        System.out.println("Tiquete eliminado exitosamente.");
    } else {
        System.out.println("Tiquete no encontrado.");
    }
}

public static void main ( String[] args ) {
    VistaConsola vista = new VistaConsola();
    vista.mostrarMenu();
}
}

```

**Controlador:** En ruta comandos a los modelos y vistas. En donde aplicamos el CRUD acrónimo que significa Create (Crear), Read (Leer), Update (Actualizar) y Delete (Borrar).

**ControladorParque:** Maneja la interacción entre las entidades y la base de datos. Incluye métodos para agregar, listar, actualizar y eliminar clientes, empleados, atracciones y tiquetes.



También gestiona la lógica de ingreso al parque y la validación de requisitos para las atracciones. Su código es el siguiente:

```
package com.salitreparquemagicoapp.controlador;

import com.salitreparquemagicoapp.db.DatabaseConnection;
import com.salitreparquemagicoapp.modelo.Atraccion;
import com.salitreparquemagicoapp.modelo.Cliente;
import com.salitreparquemagicoapp.modelo.Empleado;
import com.salitreparquemagicoapp.modelo.Estacion;
import com.salitreparquemagicoapp.modelo.Tiquete;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class ControladorParque {
    private Connection connection;

    public ControladorParque () {
        try {
            connection = DatabaseConnection.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void registrarCliente ( Cliente cliente ) {
        String sql = "INSERT INTO clientes (nombre, cedula, telefono, correo, estatura, edad, contacto_familiar, visitas, puede_recibir_promociones) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setString(1, cliente.getNombre());
            pstmt.setString(2, cliente.getCedula());
            pstmt.setString(3, cliente.getTelefono());
            pstmt.setString(4, cliente.getCorreo());
            pstmt.setDouble(5, cliente.getEstatura());
            pstmt.setInt(6, cliente.getEdad());
            pstmt.setString(7, cliente.getContactoFamiliar());
            pstmt.setInt(8, cliente.getVisitas());
            pstmt.setBoolean(9, cliente.setPuedeRecibirPromociones()); //
            // Agregar el estado de promociones
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public List<Cliente> listarClientes () {
        List<Cliente> clientes = new ArrayList<>();
        String sql = "SELECT * FROM clientes";
    }
}
```

```

        try (Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery(sql)) {
            while (rs.next()) {
                Cliente cliente = new Cliente(
                    rs.getString("nombre"),
                    rs.getString("cedula"),
                    rs.getString("telefono"),
                    rs.getString("correo"),
                    rs.getDouble("estatura"),
                    rs.getInt("edad"),
                    rs.getString("contacto_familiar"),
                    rs.getInt("visitas")
                );
                // Establecer el estado de promociones basado en la edad
                cliente.setPuedeRecibirPromociones(rs.getBoolean("puede_recibir_promociones"));

                clientes.add(cliente);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return clientes;
    }

    public void actualizarCliente ( Cliente cliente ) {
        String sql = "UPDATE clientes SET nombre = ?, telefono = ?,
correo = ?, estatura = ?, edad = ?, contacto_familiar = ?, visitas = ?,
puede_recibir_promociones = ? WHERE cedula = ?";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setString(1, cliente.getNombre());
            pstmt.setString(2, cliente.getTelefono());
            pstmt.setString(3, cliente.getCorreo());
            pstmt.setDouble(4, cliente.getEstatura());
            pstmt.setInt(5, cliente.getEdad());
            pstmt.setString(6, cliente.getContactoFamiliar());
            pstmt.setInt(7, cliente.getVisitas());
            pstmt.setBoolean(8, cliente.getPuedeRecibirPromociones()); //
            Actualizar el estado de promociones
            pstmt.setString(9, cliente.getCedula());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public boolean eliminarCliente ( String cedula ) {
        String sql = "DELETE FROM clientes WHERE cedula = ?";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setString(1, cedula);
            return pstmt.executeUpdate() > 0;
        } catch (SQLException e) {

```

```

        e.printStackTrace();
        return false;
    }
}

// Métodos para gestionar Empleados
public void registrarEmpleado ( Empleado empleado ) { String sql =
"INSERT INTO empleados (nombre, cedula, telefono, correo, cargo,
horario_laboral) VALUES (?, ?, ?, ?, ?, ?)";
    try (PreparedStatement pstmt = connection.prepareStatement(sql))
    { pstmt.setString(1, empleado.getNombre()); pstmt.setString(2,
empleado.getCedula());
        pstmt.setString(3, empleado.getTelefono());
        pstmt.setString(4, empleado.getCorreo());
        pstmt.setString(5, empleado.getCargo());
        pstmt.setString(6, empleado.getHorarioLaboral());
        pstmt.executeUpdate();
    } catch (SQLException e) { e.printStackTrace(); } }

public List<Empleado> listarEmpleados () {
List<Empleado> empleados = new ArrayList<>();
String sql = "SELECT * FROM empleados";
    try (Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next()) {
            Empleado empleado = new Empleado(
                rs.getString("nombre"),
                rs.getString("cedula"),
                rs.getString("telefono"),
                rs.getString("correo"),
                rs.getString("cargo"),
                rs.getString("horario_laboral")
            );
            empleados.add(empleado);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return empleados;
}

public void actualizarEmpleado ( Empleado empleado ) {
    String sql = "UPDATE empleados SET nombre = ?, telefono = ?,
correo = ?, cargo = ?, horario_laboral = ? WHERE cedula = ?";
    try (PreparedStatement pstmt = connection.prepareStatement(sql))
    {
        pstmt.setString(1, empleado.getNombre());
        pstmt.setString(2, empleado.getTelefono());
        pstmt.setString(3, empleado.getCorreo());
        pstmt.setString(4, empleado.getCargo());
        pstmt.setString(5, empleado.getHorarioLaboral());
        pstmt.setString(6, empleado.getCedula());
        pstmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

```

    }
}

public boolean eliminarEmpleado ( String cedula ) {
    String sql = "DELETE FROM empleados WHERE cedula = ?";
    try (PreparedStatement pstmt = connection.prepareStatement(sql))
    {
        pstmt.setString(1, cedula);
        return pstmt.executeUpdate() > 0;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}

// Método para registrar una estación
public void registrarEstacion ( Estacion estacion ) {
    String sql = "INSERT INTO estaciones (nombre, ubicacion,
capacidad, habilitada, maquina_disponible) VALUES (?, ?, ?, ?, ?)";
    try (PreparedStatement pstmt = connection.prepareStatement(sql))
    {
        pstmt.setString(1, estacion.getNombre());
        pstmt.setString(2, estacion.getUbicacion());
        pstmt.setInt(3, estacion.getCapacidad());
        pstmt.setBoolean(4, estacion.isHabilitada());
        pstmt.setBoolean(5, estacion.isMaquinaDisponible()); //
Cambiado a isMaquinaDisponible
        pstmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

// Método para listar estaciones
public List<Estacion> listarEstaciones () {
    List<Estacion> estaciones = new ArrayList<>();
    String sql = "SELECT * FROM estaciones";
    try (Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next()) {
            Estacion estacion = new Estacion(
                rs.getString("nombre"),
                rs.getString("ubicacion"),
                rs.getInt("capacidad")
            );
            estacion.setHabilitada(rs.getBoolean("habilitada"));
            estacion.setMaquinaDisponible(rs.getBoolean("maquina_disponible")); //
Cambiado a setMaquinaDisponible
            estaciones.add(estacion);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return estaciones;
}

```

```

    }

    // Método para actualizar una estación
    public void actualizarEstacion ( Estacion estacion ) {
        String sql = "UPDATE estaciones SET ubicacion = ?, capacidad = ?,
habilitada = ?, maquina_disponible = ? WHERE nombre = ?";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setString(1, estacion.getUbicacion());
            pstmt.setInt(2, estacion.getCapacidad());
            pstmt.setBoolean(3, estacion.isHabilitada());
            pstmt.setBoolean(4, estacion.isMaquinaDisponible()); //
Cambiado a isMaquinaDisponible
            pstmt.setString(5, estacion.getNombre());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // Método para eliminar una estación
    public boolean eliminarEstacion ( String nombre ) {
        String sql = "DELETE FROM estaciones WHERE nombre = ?";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setString(1, nombre);
            return pstmt.executeUpdate() > 0;
        } catch (SQLException e) {
            e.printStackTrace();
            return false;
        }
    }

    // Método para registrar una atracción
    public void registrarAtraccion ( Atraccion atraccion ) {
        String sql = "INSERT INTO atracciones (nombre, descripcion,
costo, altura_minima, edad_minima, clasificacion, condiciones_uso,
habilitada) VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setString(1, atraccion.getNombre());
            pstmt.setString(2, atraccion.getDescripcion());
            pstmt.setDouble(3, atraccion.getCosto());
            pstmt.setDouble(4, atraccion.getAlturaMinima());
            pstmt.setInt(5, atraccion.getEdadMinima());
            pstmt.setString(6, atraccion.getClasificacion());
            pstmt.setString(7, atraccion.getCondicionesUso());
            pstmt.setBoolean(8, atraccion.isHabilitada());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

```
// Método para listar atracciones
public List<Atraccion> listarAtracciones () {
    List<Atraccion> atracciones = new ArrayList<>();
    String sql = "SELECT * FROM atracciones";
    try (Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next()) {
            Atraccion atraccion = new Atraccion(
                rs.getString("nombre"),
                rs.getString("descripcion"),
                rs.getDouble("costo"),
                rs.getDouble("altura_minima"),
                rs.getInt("edad_minima"),
                rs.getString("clasificacion"),
                rs.getString("condiciones_uso")
            );
            atraccion.setHabilitada(rs.getBoolean("habilitada")); //
            Establecer el estado de habilitación
            atracciones.add(atraccion);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return atracciones;
}

// Método para actualizar una atracción
public void actualizarAtraccion ( Atraccion atraccion ) {
    String sql = "UPDATE atracciones SET descripcion = ?, costo = ?,
altura minima = ?, edad_minima = ?, clasificacion = ?, condiciones_uso =
?, habilitada = ? WHERE nombre = ?";
    try (PreparedStatement pstmt = connection.prepareStatement(sql))
    {
        pstmt.setString(1, atraccion.getDescripcion());
        pstmt.setDouble(2, atraccion.getCosto());
        pstmt.setDouble(3, atraccion.getAlturaMinima());
        pstmt.setInt(4, atraccion.getEdadMinima());
        pstmt.setString(5, atraccion.getClasificacion());
        pstmt.setString(6, atraccion.getCondicionesUso());
        pstmt.setBoolean(7, atraccion.isHabilitada());
        pstmt.setString(8, atraccion.getNombre());
        pstmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

// Método para eliminar una atracción
public boolean eliminarAtraccion ( String nombre ) {
    String sql = "DELETE FROM atracciones WHERE nombre = ?";
    try (PreparedStatement pstmt = connection.prepareStatement(sql))
    {
        pstmt.setString(1, nombre);
        return pstmt.executeUpdate() > 0;
    }
}
```

```

    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}

// Método para registrar un tiquete
public void registrarTiquete ( Tiquete tiquete ) {
    String sql = "INSERT INTO tiquetes (id, precio, fecha_compra,
cedula_cliente, visitas_cliente, fecha_validez) VALUES (?, ?, ?, ?, ?,
?)";

    try (PreparedStatement pstmt = connection.prepareStatement(sql))
    {
        pstmt.setString(1, tiquete.getId());
        pstmt.setDouble(2, tiquete.getPrecio());
        pstmt.setDate(3, new
java.sql.Date(tiquete.getFechaCompra().getTime())); // Convertir a
java.sql.Date
        pstmt.setString(4, tiquete.getCedulaCliente()); // Agregar
cédula del cliente
        pstmt.setInt(5, tiquete.getVisitasCliente()); // Agregar
número de visitas del cliente
        pstmt.setDate(6, new
java.sql.Date(tiquete.getFechaValidez().getTime())); // Convertir a
java.sql.Date
        pstmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

// Método para listar tiquetes
public List<Tiquete> listarTiquetes () {
    List<Tiquete> tiquetes = new ArrayList<>();
    String sql = "SELECT * FROM tiquetes";
    try (Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next()) {
            Tiquete tiquete = new Tiquete(
                rs.getString("id"),
                rs.getDouble("precio"),
                rs.getDate("fecha_compra"),
                rs.getString("cedula_cliente"), // Obtener cédula
del cliente
                rs.getInt("visitas_cliente") // Obtener número de
visitas del cliente
            );
            tiquete.setFechaValidez(rs.getDate("fecha_validez")); //
Establecer fecha de validez
            tiquetes.add(tiquete);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

```

        return tiquetes;
    }

    // Método para actualizar un tiquete
    public void actualizarTiquete ( Tiquete tiquete ) {
        String sql = "UPDATE tiquetes SET precio = ?, fecha_compra = ?,
cedula_cliente = ?, visitas_cliente = ?, fecha_validez = ? WHERE id = ?";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setDouble(1, tiquete.getPrecio());
            pstmt.setDate(2, new
java.sql.Date(tiquete.getFechaCompra().getTime())); // Convertir a
java.sql.Date
            pstmt.setString(3, tiquete.getCedulaCliente()); // Actualizar
cédula del cliente
            pstmt.setInt(4, tiquete.getVisitasCliente()); // Actualizar
número de visitas del cliente
            pstmt.setDate(5, new
java.sql.Date(tiquete.getFechaValidez().getTime())); // Convertir a
java.sql.Date
            pstmt.setString(6, tiquete.getId());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // Método para eliminar un tiquete
    public boolean eliminarTiquete ( String id ) {
        String sql = "DELETE FROM tiquetes WHERE id = ?";
        try (PreparedStatement pstmt = connection.prepareStatement(sql))
        {
            pstmt.setString(1, id);
            return pstmt.executeUpdate() > 0;
        } catch (SQLException e) {
            e.printStackTrace();
            return false;
        }
    }
}

```

## Descripción de la Clase ControladorParque

### ✓ Atributos:

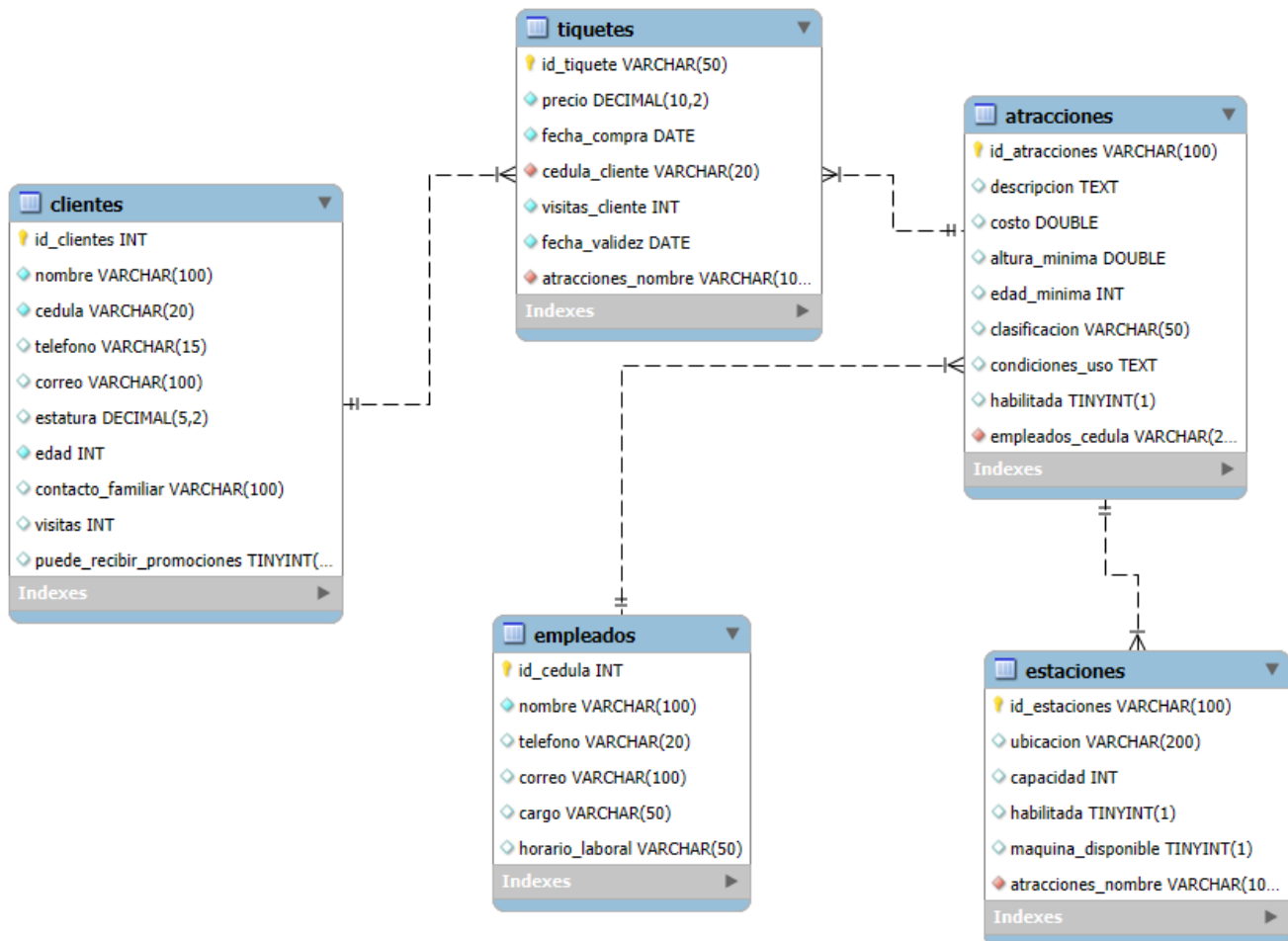
- **atracciones:** Lista de atracciones disponibles en el parque.
- **clientes:** Lista de clientes registrados en el sistema.

### ✓ empleados: Lista de empleados registrados en el sistema.



- ✓ **tiquetes:** Lista de tiquetes comprados.
- ✓ **Constructor:** Inicializa las listas de atracciones, clientes, empleados y tiquetes.
- ✓ **Métodos:**
  - **agregarAtraccion(Atraccion atraccion):** Agrega una nueva atracción al parque.
  - **registrarCliente(Cliente cliente):** Registra un nuevo cliente en el sistema.
  - **registrarEmpleado(Empleado empleado):** Registra un nuevo empleado en el sistema.
  - **comprarTiquete(String id, double costo, boolean esFrecuente):** Permite a un cliente comprar un tiquete y lo agrega a la lista de tiquetes.
  - **calcularFechaValidez():** Calcula la fecha de validez del tiquete, que es un día a partir de la compra.
  - **verificarIngreso(Cliente cliente, Atraccion atraccion):** Verifica si un cliente puede ingresar a una atracción según sus requisitos.
  - **listarAtracciones():** Muestra todas las atracciones disponibles en el parque.
  - **listarClientes():** Muestra todos los clientes registrados en el sistema.
  - **listarEmpleados():** Muestra todos los empleados registrados en el sistema.
  - **habilitarAtraccion(Atraccion atraccion, boolean habilitada):** Habilita o deshabilita una atracción según el parámetro proporcionado.
- **Clase principal:**
  - **ParqueSalitreMagicoApp:** Punto de entrada de la aplicación que inicializa el controlador y la vista.
- **Paquete DB:** Contiene la clase para manejar la conexión a la base de datos.
  - **DatabaseConnection:** Establece la conexión con la base de datos MySQL.

## EER Diagram



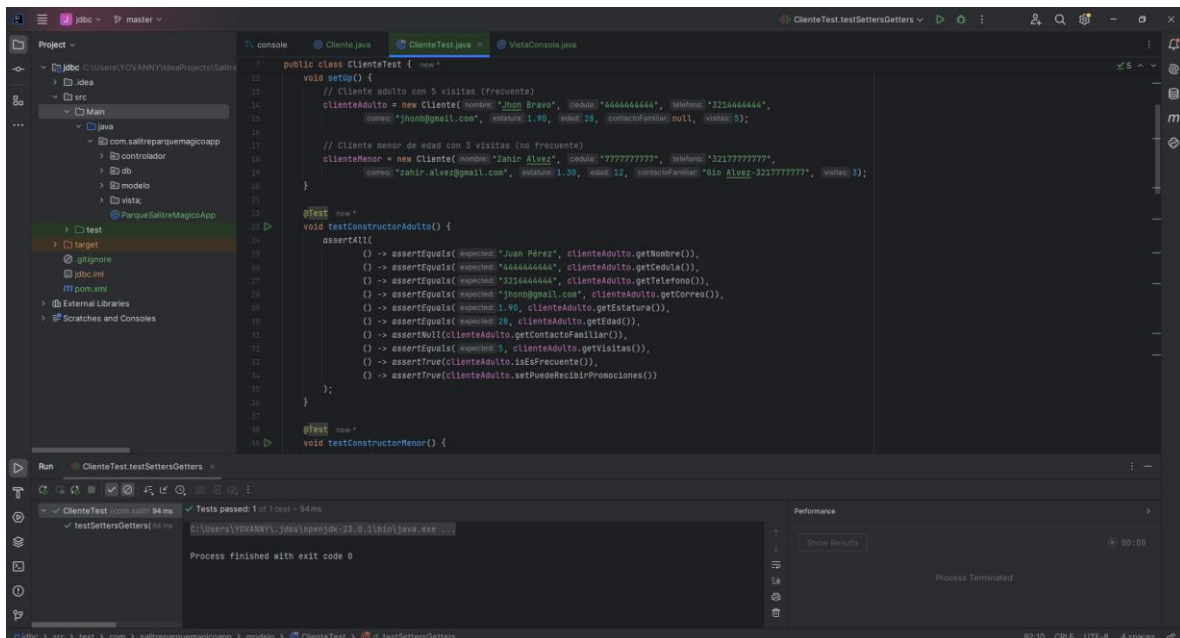
### Explicación de las relaciones:

1. CLIENTE - TIQUETE (1:N)
  - Un cliente puede comprar múltiples tiquetes
  - Un tiquete pertenece a un único cliente
  - Relación mediante id\_cliente en tabla tiquetes
2. TIQUETE - ATRACCION (N:1)
  - Un tiquete da acceso a una atracción
  - Una atracción puede ser accedida por varios tiquetes
  - Relación mediante id\_atraccion en tabla tiquetes
3. EMPLEADO - ATRACCIONES (1:N)
  - Un empleado puede operar varias atracciones
  - Una atracción es operada por un empleado a la vez

- Relación mediante id\_empleado en tabla atracciones
4. ATRACCION - ESTACIONES (1:N)
- Una atracción puede tener múltiples estaciones
  - Una estación pertenece a una única atracción
  - Relación mediante id\_atraccion en tabla estaciones

## PRUEBAS UNITARIAS

### Cliente Test:



```
public class ClienteTest {  
    void setUp() {  
        // Cliente adulto con 5 visitas (frecuente)  
        clienteAdulto = new Cliente(nombre: "Juan Pérez", cedula: "4444444444", telefono: "3214444444",  
            correo: "john@gmail.com", visitas: 1.99, edad: 28, contactoFamiliar: null, visitas: 5);  
  
        // Cliente menor de edad con 2 visitas (no frecuente)  
        clienteMenor = new Cliente(nombre: "Zahir Alvez", cedula: "7777777777", telefono: "3217777777",  
            correo: "zahir.alvez@gmail.com", visitas: 1.39, edad: 12, contactoFamiliar: "Bio Alvez-3217777777", visitas: 5);  
    }  
  
    @Test  
    void testConstructorAdulto() {  
        assertEquals("Nombre", clienteAdulto.getNombre(), "Juan Pérez");  
        assertEquals("Cédula", clienteAdulto.getCedula(), "4444444444");  
        assertEquals("Teléfono", clienteAdulto.getTelefono(), "3214444444");  
        assertEquals("Correo", clienteAdulto.getCorreo(), "john@gmail.com");  
        assertEquals("Edad", clienteAdulto.getEdad(), 28);  
        assertEquals("Visitas", clienteAdulto.getVisitas(), 1.99);  
        assertEquals("Contacto Familiar", clienteAdulto.getContactoFamiliar(), null);  
        assertEquals("Visitas", clienteAdulto.getVisitas(), 5);  
        assertTrue(clienteAdulto.isFrecuente());  
        assertTrue(clienteAdulto.setPuedeRecibirPromociones());  
    }  
  
    @Test  
    void testConstructorMenor() {  
        assertEquals("Nombre", clienteMenor.getNombre(), "Zahir Alvez");  
        assertEquals("Cédula", clienteMenor.getCedula(), "7777777777");  
        assertEquals("Teléfono", clienteMenor.getTelefono(), "3217777777");  
        assertEquals("Correo", clienteMenor.getCorreo(), "zahir.alvez@gmail.com");  
        assertEquals("Edad", clienteMenor.getEdad(), 12);  
        assertEquals("Visitas", clienteMenor.getVisitas(), 1.39);  
        assertEquals("Contacto Familiar", clienteMenor.getContactoFamiliar(), "Bio Alvez-3217777777");  
        assertEquals("Visitas", clienteMenor.getVisitas(), 5);  
        assertFalse(clienteMenor.isFrecuente());  
        assertFalse(clienteMenor.setPuedeRecibirPromociones());  
    }  
}
```

Estas pruebas unitarias cubren los siguientes aspectos principales:

- ✓ Configuración inicial (setUp):
  - Crea dos clientes de prueba: un adulto frecuente y un menor no frecuente
- ✓ Pruebas del constructor:
  - Verifica la correcta inicialización para clientes adultos y menores
  - Comprueba la lógica de contacto familiar y promociones según la edad
- ✓ Pruebas de lógica de negocio:
  - Actualización del contacto familiar al cambiar la edad
  - Actualización del estado de cliente frecuente

Cálculo de descuentos según número de visitas

Verificación de requisitos de estatura

✓ Pruebas de getters y setters:

Verifica el funcionamiento correcto de todos los métodos de acceso

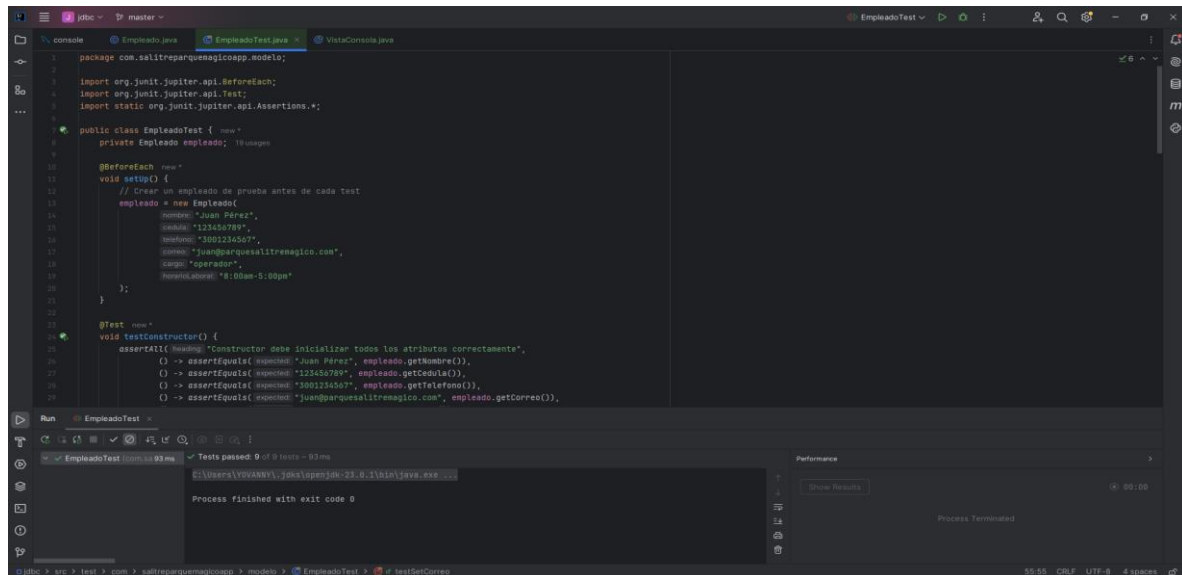
✓ Las pruebas utilizan las siguientes características de JUnit:

@BeforeEach: Para inicializar los objetos de prueba

assertAll: Para agrupar múltiples aserciones relacionadas

✓ Diferentes tipos de aserciones: assertEquals, assertTrue, assertFalse, assertNull

### ➤ Empleado Test:



```
package com.salitreparquesmagicapp.modelo;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class EmpleadoTest {
    private Empleado empleado;

    @BeforeEach
    void setup() {
        // Crear un empleado de prueba antes de cada test
        empleado = new Empleado(
            nombre: "Juan Pérez",
            email: "123456789",
            telefono: "3001234567",
            correo: "juan@parquesalitemagico.com",
            cargo: "operador",
            horario: "8:00am-5:00pm"
        );
    }

    @Test
    void testConstructor() {
        assertEquals("Constructor debe inicializar todos los atributos correctamente",
            () -> assertEquals(nombre: "Juan Pérez", empleado.getNombre()),
            () -> assertEquals(email: "123456789", empleado.getEmail()),
            () -> assertEquals(telefono: "3001234567", empleado.getTelefono()),
            () -> assertEquals(correo: "juan@parquesalitemagico.com", empleado.getCorreo()),
            () -> assertEquals(cargo: "operador", empleado.getCargo()),
            () -> assertEquals(horario: "8:00am-5:00pm", empleado.getHorario()));
    }
}
```

Run EmpleadoTest

Tests passed: 9 of 9 tests - 93ms

Process finished with exit code 0

Estas pruebas unitarias se centra en:

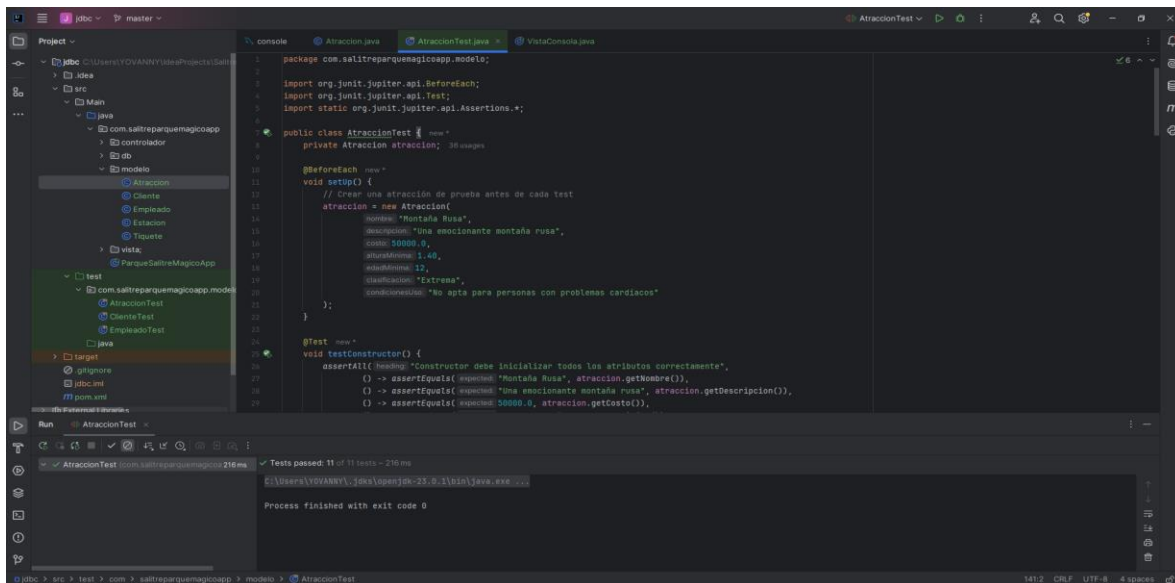
1. Setup inicial:
  - ✓ Crea un empleado de prueba antes de cada test
2. Pruebas del constructor:
  - ✓ Verifica que todos los atributos se inicialicen correctamente
3. Pruebas de los setters y getters:
  - ✓ Prueba individual para cada método set/get
  - ✓ Verifica que los valores se actualicen correctamente
4. Pruebas de diferentes tipos de empleados:
  - ✓ Verifica la creación de empleados con diferentes cargos
  - ✓ Comprueba que los cargos se asignen correctamente
5. Pruebas de casos límite:

- ✓ Verifica el comportamiento con strings vacíos

Las pruebas cubren la funcionalidad básica de la clase Empleado sin depender de otras clases del sistema, enfocándose en:

- Creación de empleados
- Modificación de atributos
- Recuperación de información
- Diferentes tipos de cargos

## ➤ Atracción Test:



```

package com.salitreparquemagico.modelo;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class AtraccionTest {
    private Atraccion atraccion;

    @BeforeEach
    void setUp() {
        // Crear una atracción de prueba antes de cada test
        atraccion = new Atraccion(
            nombre: "Montaña Rusa",
            descripcion: "Una emocionante montaña rusa",
            costo: 50000.0,
            habilitada: true,
            contadorVisitas: 12,
            clasificacion: "Extrema",
            condiciones: "No apta para personas con problemas cardiacos"
        );
    }

    @Test
    void testConstructor() {
        // Crear una atracción de prueba antes de cada test
        atraccion = new Atraccion(
            nombre: "Montaña Rusa",
            descripcion: "Una emocionante montaña rusa",
            costo: 50000.0,
            habilitada: true,
            contadorVisitas: 12,
            clasificacion: "Extrema",
            condiciones: "No apta para personas con problemas cardiacos"
        );
    }
}

```

Estas pruebas unitarias cubre:

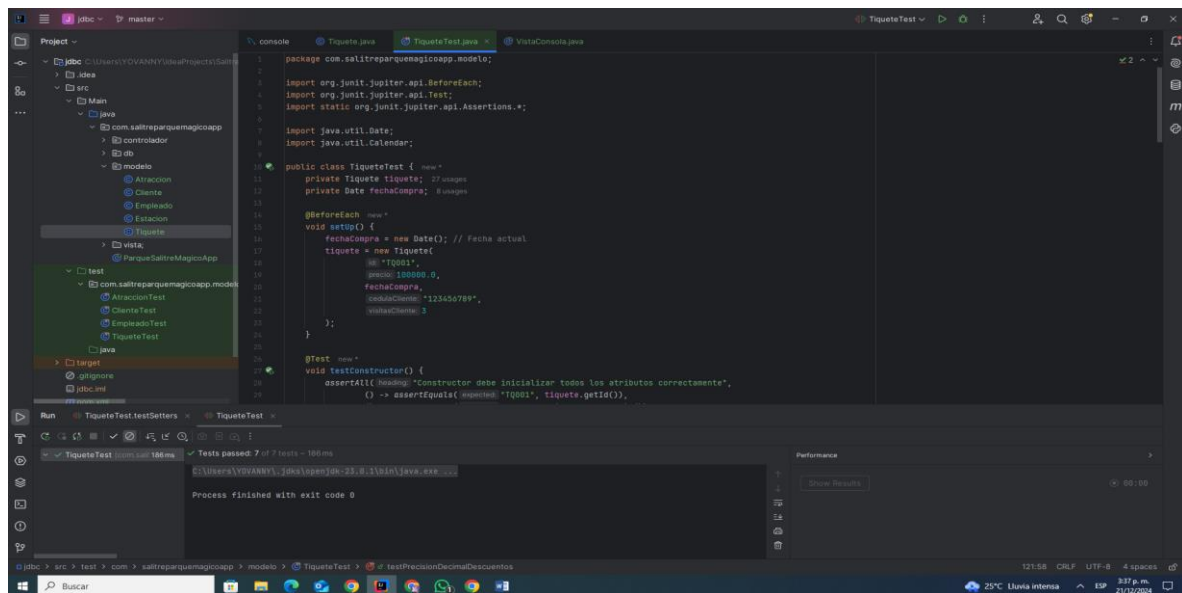
1. Configuración inicial (setUp):
  - ✓ Crea una atracción de prueba antes de cada test
2. Pruebas del constructor:
  - ✓ Verifica la inicialización correcta de todos los atributos
  - ✓ Comprueba los valores por defecto (habilitada = true, contadorVisitas = 0)
3. Pruebas de setters y getters:
  - ✓ Prueba individual para cada método set/get
  - ✓ Verifica que los valores se actualicen correctamente
4. Pruebas del estado de habilitación:
  - ✓ Verifica el estado por defecto
  - ✓ Comprueba la habilitación/deshabilitación
5. Pruebas del contador de visitas:
  - ✓ Verifica el valor inicial

- ✓ Comprueba el incremento correcto
- 6. Pruebas de diferentes tipos de atracciones:
  - ✓ Verifica la creación de atracciones con diferentes características
  - ✓ Comprueba las relaciones lógicas entre diferentes tipos de atracciones

Las pruebas aseguran que:

- La creación de atracciones funciona correctamente
- Los atributos se pueden modificar apropiadamente
- El contador de visitas funciona como se espera
- El estado de habilitación se maneja correctamente
- Se pueden crear diferentes tipos de atracciones con características distintas

## ➤ Tiquete Test:



Estas pruebas cubre:

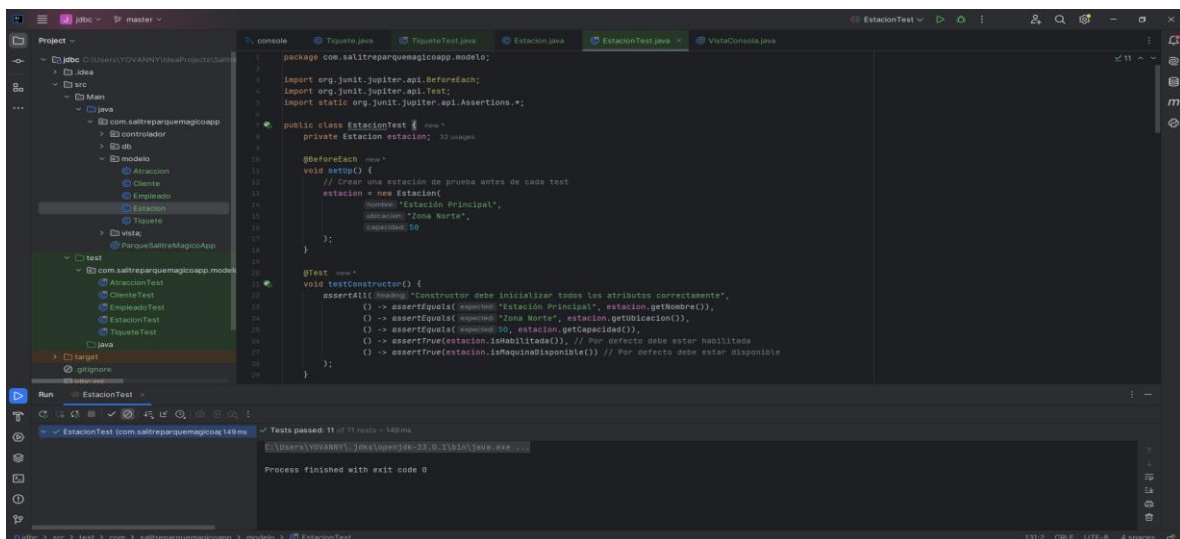
1. Configuración inicial (setUp):
  - ✓ Crea un tiquete de prueba antes de cada test
  - ✓ Inicializa la fecha de compra
2. Pruebas del constructor:
  - ✓ Verifica la inicialización correcta de todos los atributos
  - ✓ Comprueba los valores calculados (esFrecuente, fechaValidez)
3. Pruebas de cálculos de fechas:
  - ✓ Verifica el cálculo correcto de la fecha de validez
  - ✓ Prueba la validación de tiquetes vigentes y vencidos
4. Pruebas de cálculos de precios:

- ✓ Verifica diferentes niveles de descuento
- ✓ Prueba la precisión en cálculos decimales
- 5. Pruebas de lógica de cliente frecuente:
  - ✓ Verifica la actualización del estado frecuente
  - ✓ Prueba los umbrales de visitas
- 6. Pruebas de setters y getters:
  - ✓ Verifica la correcta modificación de todos los atributos

Las pruebas aseguran que:

- Los tickets se crean correctamente
- Los descuentos se calculan con precisión
- Las fechas de validez son correctas
- El estado de cliente frecuente se actualiza apropiadamente
- Los setters y getters funcionan como se espera

### ➤ Estación Test



The screenshot shows an IDE with the following components:

- Project Explorer:** Shows the project structure with packages like `com.salitreparquemagicapp.modelo` and `com.salitreparquemagicapp.modelo.test`. The `EstacionTest` class is highlighted.
- Editor:** Displays the `EstacionTest.java` file. The code includes imports for JUnit, a `@BeforeEach` method to create a test station, and a `@Test` method `testConstructor()` that verifies the station's attributes using `assertEquals` and `assertTrue`.
- Run Console:** Shows the execution results of the test. It indicates that 11 tests passed out of 11, with a total time of 149 ms. The process finished with exit code 0.

Estas pruebas cubre:

1. Configuración inicial (setUp):
  - ✓ Crea una estación de prueba antes de cada test
2. Pruebas del constructor:
  - ✓ Verifica la inicialización correcta de todos los atributos
  - ✓ Comprueba los valores por defecto (habilitada y máquina disponible)
3. Pruebas de setters y getters:
  - ✓ Prueba individual para cada método set/get
  - ✓ Verifica que los valores se actualicen correctamente

4. Pruebas de estados:
  - ✓ Verifica los cambios de estado de habilitación
  - ✓ Comprueba los cambios de estado de la máquina
5. Pruebas de casos especiales:
  - ✓ Prueba con capacidad negativa
  - ✓ Prueba con capacidad cero
  - ✓ Prueba con strings vacíos
  - ✓ Prueba de modificación completa de todos los atributos
6. Pruebas de diferentes configuraciones:
  - ✓ Verifica la creación de estaciones con diferentes capacidades y ubicaciones

Las pruebas aseguran que:

- Las estaciones se crean correctamente
- Los atributos se pueden modificar apropiadamente
- Los estados se manejan correctamente
- Se pueden crear diferentes tipos de estaciones
- Se manejan correctamente los casos límite